Geometric Transformations on Images

Interpolation, Affine Transformations, Imaging, Re-Projection and Pseudo-3D

Lab Exercise 1 for the first 2 weeks of Beeldverwerken

Informatics Institute
University of Amsterdam
The Netherlands

March 24, 2016

1 Introduction

In many image processing applications you have access to a number of image transformation methods. These include rotating, stretching and even shearing an image.

These methods can be used to create some artistic effect, but they are also useful in correcting things like camera rotation, perspective distortions, etc, that were introduced when the picture was taken.

The input for these methods is always a quadrilateral region of a discrete image F. That means that you only know the intensity values at discrete integral pixel positions. The output consists of a rectangular image G with axes aligned with the Cartesian coordinate axes x and y.

In an arbitrary image transformation, there is no guarantee that an “input”-pixel will be positioned at a pixel in the “output” image as well. Rather, most of the time your output image pixels will “look at” image positions in the input image, which are “between” pixels.

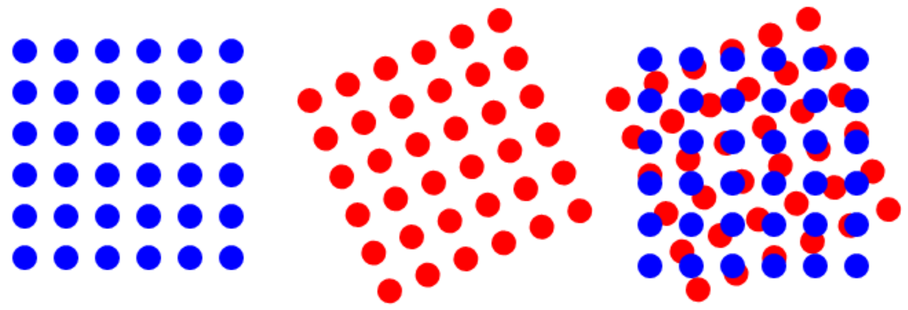

Figure 1: Pixel positions in an original and a transformed version of an image don’t always agree.

So you need access to intensity values which are not on the sampling grid of the original image, *e.g.* the intensitiy value at position (6.4, 7.3).

In this Exercise you will examine and implement interpolation techniques which solve this problem. Then you will implement image transformations employing these techniques, such as rotations and projective transformations. This requires determining the desired transformation from the image data or from user input. We will implement the latter.

The Matlab code that is given throughout the exercise is to help you structuring your program. It is not obligatory to use these exact functions, but it will make your life easier when we later re-use the code from this assignment to solve real computer vision problems.

2 Interpolation

The transformation algorithm needs a way to access the original image F in locations that are not on the sampling grid. We thus need a function pixelValue that returns the value in the location (x, y) even for non-integer coordinate values.

Theory Questions

2.1. **(2 points)** Nearest-neighbour interpolation is a very simple method for doing interpolation. Suppose we have a one dimensional function $F(x)$, that is only defined when x is an integer (it was *sampled* at those points). Let $\text{floor}(x) = \lfloor x \rfloor$ be a function that rounds x down to the nearest integer that is smaller or equal to x. Use the floor function and $F(x)$ to give the definition for nearest-neighbour interpolation of a point x.

2.2. **(2 points)** Linear interpolation in 1 dimension is treated in Section 2.5 of the Lecture Notes. Now let $k = \lfloor x \rfloor$. We know that $F(x)$ is defined for $F(k)$ and $F(k+1)$, so we have the points $(k, F(k))$ and $(k+1, F(k+1))$. If we want to fit a line $f_1(x) = ax + b$ through both these points, which two equations should we solve?

2.3. **(2 points)** Write these equations in standard linear algebra form as a matrix equation, and solve to find the result (explaining the "It leads to" in the Notes).

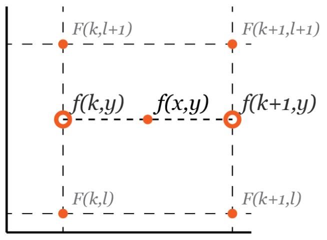

Figure 2: Bilinear interpolation

2.4. **(2 points)** You can apply the previous method for both dimensions independently, first yielding the two open circles in Figure 2, then interpolating between those two to get the value for $f_1(x, y)$. Do this, and verify that you end up with equation 2.3, given in the Lecture Notes and repeated below.

$$\begin{aligned} f_1(x,y) = &(1-\alpha)(1-\beta)F(k,\ell)+ \\ &(1-\alpha)\beta F(k,\ell+1)+ \\ &\alpha\beta F(k+1,\ell+1)+ \\ &\alpha(1-\beta)F(k+1,\ell) \end{aligned}$$

2.1 Questions and Exercises

⋆ **(2 points)** Write a Matlab function `pixelValue` that returns the value of a pixel at real valued coordinates `(x,y)`. The function should look like:

Listing 1: Image Interpolation

```
function color = pixelValue( image, x, y, method )
% pixel value at real coordinates
if inImage(size(image),x,y)
  % do the interpolation
  switch(method)
    case 'nearest'
      % Do nearest neighbour
      return;
    case 'linear'
       % Do bilinear interpolation
  end %end switch
else
  % return a constant
end
```

⋆ **(10 points)** Implement nearest neighbour interpolation (`method = 'nearest'`) and bilinear interpolation (`method = 'linear'`), as are described in the syllabus. Take the necessary steps to deal with the "border problem" (*i.e.* implement the function inImage). The simplest choice is to return a constant value in case `(x,y)` is not within the domain of the image, but have a look at Figure 2.19 of the Lecture Notes and the accompanying text for the other options. Additionally, please note that in Matlab the image integer coordinates start at 1 not at 0 (as is common in many other computer languages).

• **(2 points)** Perform a preliminary test of your function `pixelValue` by using it in a `profile` function that samples the image in `n` equidistant points along a line from `(x0,y0)` to `(x1,y1)`. Make sure the result makes sense. The profile function is:

Listing 2: Image Profile

```
function line = profile( image, x0, y0, x1, y1, n, method )
% profile of an image along straight line in n points
x = linspace(x0, x1, n); y = linspace(y0,y1,n);
for i = 1:length(x)
  line(i) = pixelValue( image,x(i), y(i), method );
end
```

Test this function with:

```
a = imread( 'autumn.tif' );
a = im2double( rgb2gray(a) );
plot( profile( a, 100, 100, 120, 120, 100, 'linear') );
```

Experiment a bit with a varying number of points on the profile line while keeping the begin and end point fixed. With many points on the line the difference between nearest neighbor interpolation and bilinear interpolation should become clearly visible.

The easiest way to get several function plots in one figure is to 'hold' the plot (overlay a new plot on top of an earlier plot, instead of first clearing the figure).

```
reset(gcf);      % this resets the (get) current figure
hold on;         % overlay on current figure
plot( profile( a, 100, 100, 120, 120, 200, 'linear'), 'b' );
plot( profile( a, 100, 100, 120, 120, 200, 'nearest'), 'r' );
hold off;
```

The (bi)linear interpolated profile will be plotted in blue (the `'b'` option) and the nearest neighbor interpolated profile will be plotted in red (the `'r'` option).

- **(5 points)** Use different methods for the border problem (see Lecture Notes Figure 2.19 and the accompanying text) and describe their results.

3 Rotation

Now that we have two working interpolation methods, we will compare their performance both in time and quality. We will do this by rotating the image $\pi/6$ radians counterclockwise, and then $\pi/6$ radians clockwise. Then we will compare the original image with the twice rotated image using the quadratic distance measure.

Theory Questions

3.1. **(3 points)** Give the matrix $\mathbf{R}$ of a counter-clockwise rotation by ϕ radians about the origin. We will need this matrix later.

3.2. **(3 points)** How can we use this matrix to transform a point $(x, y)^T$ in the original image to a point $(x', y')^T$ in the rotated image, if we are only interested in rotations around the origin?

3.3. **(3 points)** Think of a simple way to perform a rotation about an arbitrary point $\mathbf{c}$. Hint: you will not need a different kind of rotation matrix for this.

3.4. **(3 points)** What happens to the center $\mathbf{c}$ itself if we rotate around $\mathbf{c}$? Verify your answer algebraically.

3.1 Questions and Exercises

⋆ **(15 points)** Implement a function that rotates an image through an angle ϕ counter-clockwise around its center. Notice that the origin and axes of an image in Matlab are not the same as you are used to. In Matlab, `image(x,y)` selects a point according to the axes displayed in Figure 3. (If you tilt your head to the right, you see the normal axes system.)

The function should look something like:

```
function rotatedImage = rotateImage(image, angle, method)
% Create the necessary rotation matrix

% Obtain indices needed for interpolation

% Obtain colors for the whole rotatedImage matrix
% using the specified interpolation method
```

Hint1: Some pixels may lie outside the boundaries of the original image. How do you make sure that all the pixels of the rotated image will receive a value?

Hint2: Do your initial testing with a small picture, like `cameraman.tif`. This saves a lot of time.

Hint3: Do not let the rotation function change the size of the image. If you do, evaluation of the interpolation methods might become impossible.

- **(5 points)** Matlab does not like for-loops. It is preferable to do the rotation with one matrix multiplication for all points in the image. Do this. You may include this version only and leave out the previous one from the hand-in. You will probably need the Matlab functions `reshape` and `repmat`.

  Hint: Reshaping a $N \times M$-matrix into a $1 \times (NM)$-matrix and looping over that will *not* do the trick, because Matlab will still create N times M lines of code, which have to be interpreted.

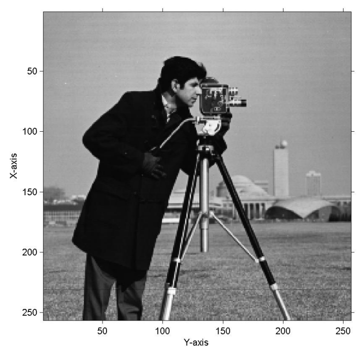

Figure 3: Image of the cameraman with the axes used by Matlab. Accessing the matrix `image` through `image(x,y)` will access a pixelcolor value according to these axes.

- **(10 points)** If you rotate the image $\pi/6$ radians counterclockwise, you will notice that part of the image will fall off the screen. Rotating it back will result in an incomplete picture, thus making it impossible to do a good evaluation. The easiest solution to this problem is creating a (black) border around the picture to prevent the important parts of the image from rotating out of the view. Do this intelligently.

- **(3 points)** Compare both interpolation methods on their processing time. You can use the Matlab function `tic` to reset the timer. Calling `toc` will return the amount of time, that has elapsed since the last `tic` command.

- **(8 points)** To compare the quality of both interpolation methods mathematically, you will need some kind of distance measure to calculate how "far" from the original the interpolated version is.

  You can assign a distance to two images (of equal size) by measuring the difference between them in each pixel and *merging* those differences into one single scalar value. Do this in a quadratic manner. This results in a distance measure that is always positive and emphasizes large differences over small ones.

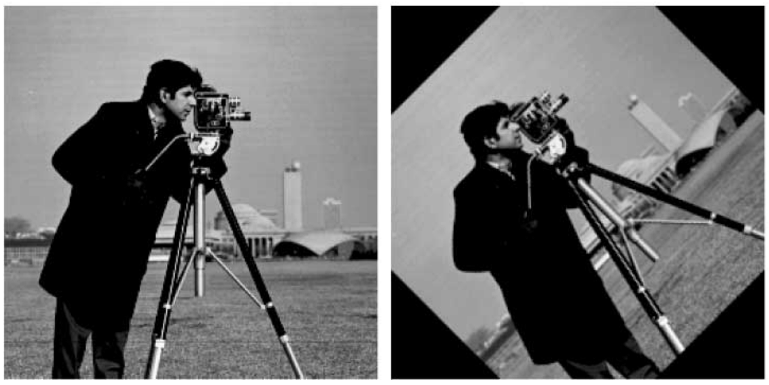

Figure 4: **Rotation.** Applying a 45 degree rotation to the cameraman image.

Describe how you calculate that distance and apply it to each pair of a double rotated image and an original image (which should look the same at first glance). Are the results of this mathematical check what you would have expected considering your perceived quality of the images? Does the black border around the picture have any influence on this distance measure?

4 Affine Transformations

Another transformation, which is more general than a rotation, is an *affine transformation.* You will use it to transform a parallelogram (a quadrilateral region in the input image) into another parallelogram (the rectangular "canvas" of the output image).

The generic formula for affinely transforming a 2D point $(x, y)^T$ into a point $(x', y')^T$ is:

$$\begin{aligned} x' &= ax + by + c \\ y' &= dx + ey + f \end{aligned} \tag{1}$$

We will take a closer look at affine transformations in general, before we move on to the main exercise.

Theory Questions

4.1. **(2 points)** What problems do you run into when trying to write (1) as a matrix equation?

4.2. **(2 points)** Explain how we can use homogeneous coordinates to perform an affine transformation as a matrix multiplication. Give the matrix multiplication that performs the transformation in (1).

4.3. **(2 points)** Consider the transformation in (1), where we replace a, b, d and e by the elements of the 2D rotation matrix. Write this down in matrix form, and note that it has three parameters: c, f and ϕ. What transformation does it perform, and what does each parameter do?

Now that we have a better understanding of affine transformations and homogeneous coordinates, we will look at the affine transformation depicted in figure 5. Suppose you want to map the

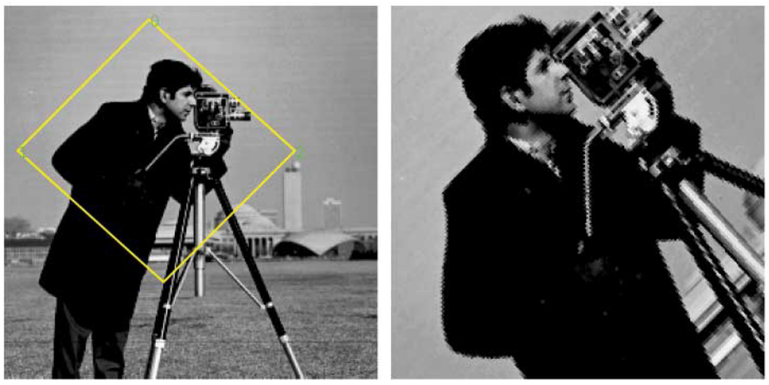

Figure 5: **Affine Transformation.** The parallelogram on the left is mapped to the rectangular image on the right.

selected parallelogram in the cameraman image onto a rectangular image "canvas". Of course, every point inside the parallelogram will be transformed by the *same* affine transformation.

So, solving for parameters a through f gives you all the information you need to perform the mapping. If you know one point and the coordinates, it is mapped to, (1) gives you two equations. You will need two more known point-mappings, if you want to solve for all the parameters. We can write these equations down in matrix form as follows.

$$\begin{pmatrix} x'_1 & x'_2 & x'_3 \\ y'_1 & y'_2 & y'_3 \end{pmatrix} = \begin{pmatrix} a & b & c \\ d & e & f \end{pmatrix} \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix} \tag{2}$$

Theory Questions

4.4. **(2 points)** To solve equation (2), we need three points in the original image and the points they are mapped to in the output image. Think of an easy way to find three such point-pairs. What are the coordinates of the output points?

4.5. **(2 points)** Why do we need three points? Why couldn't we do with less, and why don't we use more?

Solving for the affine matrix in of (2) (let's call that X) is easy in Matlab:

```
A = [x1, x2, x3; y1, y2, y3; 1, 1, 1];
B = [xx1, xx2, xx3; yy1, yy2, yy3];
X = B / A;
```

We could have written `X = B * inv(A)`. Although that would work, it is almost never a good idea to solve a set of equations using a matrix inverse calculation. Matlab "knows" this and has the special form `B / A` to solve the equation $XA = B$. Check the Matlab Help on the functions `mldivide` and `mrdivide` for details.

A more readable way of writing equation (2) is:

```
A = [x1, y1, 1; x2, y2, 1; x3, y3, 1]';
B = [xx1, yy1; xx2, yy2; xx3, yy3]';
X = B / A;
```

Can you see why?

Given the matrix X, a function `myAffine` can be written that does the actual affine transformation.

Listing 3: Affine Transformation

```
function r = myAffine(image, x1, y1, x2, y2, x3, y3, M, N, method)
  r = zeros(N, M); % allocate new image of correct size
  % calculate X (insert code for this)
  for xa = 1:M
    for ya = 1:N
      % calculate x and y (insert code for this)
      r(ya, xa) = pixelValue(image, x, y, method);
    end
  end
```

Notice how you have to take care about the different notation of image- and matrix-coordinates.

In the development of your `myAffine` function, it might be useful to be able to plot the selected parallelogram "on top of" an image. You can use the following code.

Listing 4: Plot Parallelogram

```
% plot a parallelogram overlayed on the current axis
% coordinates given are image coordinates
function plotParallelogram(x1, y1, x2, y2, x3, y3)
  hold on;
  plot([x1, x2, x3, x3-x2+x1, x1], [y1, y2, y3, y1-y2+y3, y1], ...
  'y', 'LineWidth', 2);
  text(x1, y1, '1', 'Color', [0, 1, 0], 'FontSize', 18);
  text(x2, y2, '2', 'Color', [0, 1, 0], 'FontSize', 18);
  text(x3, y3, '3', 'Color', [0, 1, 0], 'FontSize', 18);
```

This function plots on top of the figure displayed in the currently active window. You can make a figure window active using the `figure(n)` function, where `n` is the number of the figure.

4.1 Questions and Exercises

- ⋆ **(10 points)** Complete the function `myAffine` from Listing 3.
- ⋆ **(10 points)** Use your function `myAffine` to rotate the image `cameraman.tif` about its center through 45 degrees counter-clockwise. The resulting image should look like the one in Figure 4. Plot the resulting image in your report together with the exact parameters in the call to `myAffine`.
- **(2 points)** The function as given uses two nested for-loops. This is very inefficient in Matlab. Figure out a way to get the calculation of x and y out of the for-loops and eliminate one for-loop. It's ok to include only the efficient version in your hand-in.

  Hint: Again, reshaping a $N \times M$-matrix into a $1 \times (NM)$-matrix and looping over that will *not* do the trick.

5 Re-projecting Images

In general we are not so lucky that the transformation, which a camera performs on a flat piece of paper, is *affine*. That only happens if the two are relatively far away from each other (in terms of the camera's focal length).

You may have noticed that in (2) we used homogeneous coordinates on the points of the input image, but not on the output points. Even though you *could* have "enhanced" the output coordinates by an additional entry 1 afterwards, the result would have been the same. However, the most general transformation which you can apply to homogeneous coordinates in a plane, is a *projective transformation.*

This is exactly the transformation that a pinhole camera applies to a planar object when (ahem) projecting it to the image plane. In Chapter 3 you can find out how this transformation is connected to the imaging process in a camera, and we'll do an exercise about that in Section 6. For now, all you need to know is that *a projective transformation is capable of transforming an arbitrary quadrilateral in the plane into another arbitrary quadrilateral* (this could be called its 'defining property'). In particular you can use it to straighten out the picture of a piece of paper lying on a desk as in Figure 6.

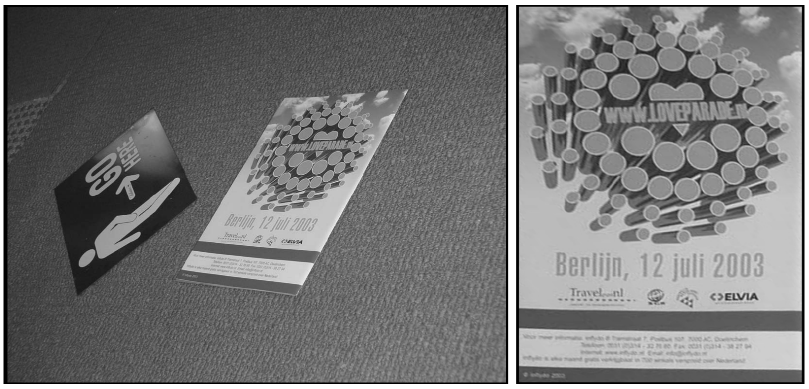

Figure 6: **Projection.** After applying the transformation, we have a frontal view of the flyer.

The equations governing the projective mapping of the coordinates of a 2D point from $(x, y)^T$ to $(x', y')^T$ look like this:

$$\begin{aligned} x' &= (ax + by + c)/(kx + ly + m) \qquad (3) \\ y' &= (dx + ey + f)/(kx + ly + m) \qquad (4) \end{aligned}$$

Two things are noticeable about these equations. The first one is the common denominator of x' and y'. Let's call that $\lambda = kx + ly + m$. The second thing is that λ depends on x and y, which, in principle, makes the equations nonlinear.

But as you have read in the 'extra' chapter on Homogeneous Coordinates, we can fortunately amend that by employing homogeneous coordinates again.

$$\begin{pmatrix} a & b & c \\ d & e & f \\ k & l & m \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} ax + by + c \\ dx + ey + f \\ kx + ly + m \end{pmatrix} = \begin{pmatrix} \lambda x' \\ \lambda y' \\ \lambda \end{pmatrix} \qquad (5)$$

Dividing the resulting three-dimensional vector by λ, the last coordinate, you arrive at the homogeneous representation of the 2D output point $(x', y', 1)^T$. That is the only additional rule to incorporate projective transformations: *interpretation of homogeneous vectors to point locations involves division by the last coordinate.*

Equation (5) is linear! So thanks to the interpretation convention of homogeneous vectors, we can use linear algebra again for our internal computations, even for this seemingly nonlinear problem! Yippee! (Seriously, though, whenever this happens it should come as a relief, since you get so much for free, both conceptually and implementationally, when you can map to linear algebra.)

The matrix on the far left of (5) is called the *projection matrix*. We denote it by M and its entries m_{11} through m_{33} (since we're beginning to run out of letters, and to show its structure).

As with affine transformations, the projection matrix M can be determined from a number of known point transformations. You can specify the minimum number of such correspondences and solve for parameters m_{11} through m_{33}, exactly. The basic technique is given in Section 3 Projective Transformations of the 'extra' chapter on Homogeneous Coordinates on BB. Read that and answer the following questions to check whether you understood that material:

Theory Questions

5.1. **(1 points)** Write (5) in terms of the m_{ij}.

5.2. **(1 points)** Work out the equations which a desired point correspondence gives you. How many are there for each point correspondence?

5.3. **(1 points)** We stated above as 'defining property' that the projective transformation can transform any arbitrary quadrilateral to another arbitrary quadrilateral. If this is the case, how many parameters should it at least have?

5.4. **(1 points)** Seemingly, there are 9 unknown m_{ij} in the 3×3 projection matrix M. Actually, there are only 8, why?

5.5. **(1 points)** How many point correspondences do you therefore minimally need to determine a projective transformation in 2D? (Bonus: and how many in 3D?)

5.6. **(1 points)** What are the unknowns? Collect those in a vector.

5.7. **(1 points)** Rewrite the equation as a matrix equation for the unknown vector. (If this succeeds, you are lucky enough to have a problem that is actually solvable using the trusted tool of linear algebra, so it is worth trying!)

5.8. **(1 points)** You do succeed (if not, try again!), and the equation can be brought into the form $A\mathbf{x} = \mathbf{0}$. The trivial solution is $\mathbf{x} = 0$. How do you avoid it? (**Hint:** Perhaps try to answer the next question first.)

5.9. **(1 points)** A general linear transformation in 3D has the $3 \times 3 = 9$ parameters of its matrix coefficients. Why is the projective transformation actually sufficiently determined by fewer parameters?

5.10. **(1 points)** Matlab can determine the kernel of a matrix A by the command 'null(A)'. How is this useful in the current problem?

Solving the resulting equation (which is eq. 5 in the Homogeneous Coordinates writeup) in Matlab is not difficult. The hardest part is constructing the matrix A. To do so, we define two matrices, `XY` and `UV`, as follows. Let the points (x_k, y_k) be collected in Matlab array `XY` such that `XY(k, 1)` equals x_k and `XY(k, 2)` equals y_k, *i.e.* the k-th row of `XY` is the point (x_k, y_k). Equivalently let `UV` store the points (x'_k, y'_k). Then construct A like this:

Listing 5: Constructing matrix A

```
x = xy(:, 1);
y = xy(:, 2);
% we cannot use x' and y' in Matlab because ' means transposed
u = uv(:, 1);
v = uv(:, 2);
o = ones(size(x));
z = zeros(size(x));
Aoddrows = [x, y, o , z, z, z, -u.*x, -u.*y, -u];
Aevenrows = [z, z, z, x, y, o, -v.*x, -v.*y, -v];
A = [Aoddrows; Aevenrows];
```

You should notice that the matrix A constructed this way is *not* exactly the one from eq.(5) of the Homogeneous Coordinates writeup. What have we done in terms of the equations? Why does this not change the solution of the homogeneous system?

5.1 Questions and Exercises

- ⋆ **(2 points)** Create a function `createProjectionMatrix` which looks like this:

  Listing 6: createProjectionMatrix

  ```
  function projMatrix = createProjectionMatrix(xy, uv)
  % calculation of projection matrix
  ```

  Is `projMatrix` the same projection matrix as the M calculated above?

- ⋆ **(10 points)** Create a function `myProjection` that takes an original image `image` and four points `x1, y1, x2, y2, x3, y3, x4, y4`, specifying a quadrilateral, and outputs the $m \times n$ rectangular image `projection`. The function could look like this:

  Listing 7: myProjection

  ```
  function projection = myProjection(image, x1, y1, x2, y2, ...
        x3, y3, x4, y4, m, n, method)
  projection = zeros(n, m); % allocate new image of correct size
  % calculate projection matrix

  for xIndex = 1:m
    for yIndex = 1:n
      %calculate x and y
      projection(yIndex, xIndex) = ...
        pixelValue(image, x, y, method);
    end
  end
  ```

  Note that it might be easier to use a 2×4 matrix `points` instead of `x1, y1, ..., x4, y4`.

- ⋆ **(2 points)** Use the function `myProjection` to get a frontal view of one of the objects in `unstraightened.jpg` (see Figure 6).

  Hint: You can use `ginput(4)` to obtain the four corner points of the object via mouse-input. See the Matlab Help on `ginput`.

- **(2 points)** The function as given uses two nested for-loops. This is very inefficient in Matlab. Figure out a way to get the calculation of x and y out of the for-loops and eliminate one for-loop. It's ok to include only the efficient version in your hand-in.

  Hint: Again, reshaping a $N \times M$-matrix into a $1 \times (NM)$-matrix and looping over that will *not* do the trick.

6 Pinhole Camera Model

In this section we briefly indicate what would be required to create your own camera model and use it to "look at" a three-dimensional scene from a general position in space, *i.e.* project a 3D scene to a 2D image.

As the term "project" suggests, this transformation is a *projective* transformation. In Chapter 5 you have already seen how a projective transformation can be made into a *linear* transformation by using homogeneous coordinates. But there you only (re-)projected 2D points to 2D points.

To treat 3D in a similar way, we generalize the trick: we introduce 4D homogeneous coordinates in space, and then 3D homogeneous coordinates in the 2D image, and map between them. So, let us codify how a 3D point with *world coordinates* (X, Y, Z) is depicted on the image in the point with *image coordinates* (x, y). The relation is:

$$\begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \sim \begin{pmatrix} \lambda x \\ \lambda y \\ \lambda \end{pmatrix} = M \begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix}$$

where M is a 3×4-matrix.

M is called the *projection matrix* or *camera matrix* (or even just the *camera*). Chapter 3 shows that it can be decomposed into

$$M = M_{\text{int}}\, M_{\text{ext}},$$

which shows that M depends on certain internal (such as the focal length, skewness of pixels etc.) and external (position and orientation in space) camera parameters. When you write code, it makes sense to maintain M_{int} and M_{ext} separately. In this way you can adjust one without changing the other. The meaning of the entries is specified in Chapter 3.

Before we create the code that simulates a pinhole camera, we will elaborate on the camera model a little more.

- In a real pinhole camera the light coming from the scene first passes the camera center and then falls on the image plane (nowadays usually a CCD chip). This flips the image upside-down and left-to-right. The camera software compensates for this effect by flipping the image back before displaying it in the view-finder (nowadays usually a TFT monitor).

  In Computer Vision one usually assumes that the image plane is *in front of* the center of projection (*i.e.* the pinhole). Is this essentially different? If not, why is it convenient?

- The standard camera has its center of projection at the (world) origin and looks in the z-direction. That is how the camera parameters are given in all literature. The cameras you are going to use will be in general position, so they should be translated and rotated relative to that, by using an appropriate M_{ext}.

- We will only vary the focal length f of the camera and not play with the other parameters in M_{int}. If you would include those, what kind of transformation would be performed on the image? **Hint:** We've done it before.
- The scene you will project consists of a wireframe cube centered on the world origin with edge-length 1 and edges parallel to the cartesian coordinate axes. If you haven't implemented it yourself you can use the functions `createCube.m` and `subPlotFaces.m`, respectively, to create a cube with a given center and edge length and to plot a set of 2D polygons (e.g. the cube's faces after the projection). Study the functions carefully before using them in your implementation!

Figure 7 gives you an example of a general camera and the view it produces.

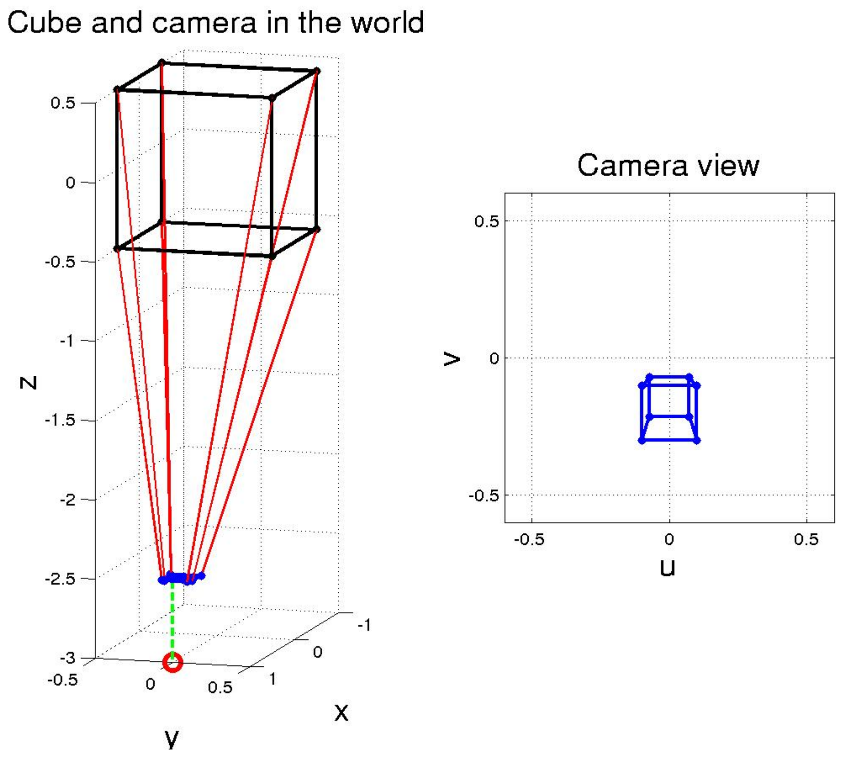

Figure 7: left: the world with camera and object, right: the camera image.

6.1 Questions and Exercises

⋆ **(0 points)** Write generic code to solve these and similar problems of seeing this cube, taking as parameters the rotation axis, angle, translation vector and focal length. In processing the points, you should make sure that they have real images on the screen of the pinhole camera. Your code should run properly on any other similar example!

Note that the cube is a wire frame, so you do not need to compute whether some faces would block the visibility of the points behind them; you can learn how to do that cleverly in a course on computer graphics.

Show the camera image for a camera which is:

- Non-rotated, pinhole at $(0, 0, -10)$, $f = 1$.
- Same, with $f = 2$. Compare with the first!
- Same, with $f = 0.5$. Compare with the previous two!
- Camera rotated 45 degrees around the axis $(1, 1, 1)$, at $(-5, -5, 0)$, with $f = 1$.

- **(0 points)** Can you check whether a point is in front of or behind the camera, and only draw visible parts of the object? Try the two following camera positions:
  - Camera at location $(0.3, 0, 0)$, $f = 0.5$. *(Tricky!)*
  - Camera at location $(-0.5, 0, 0)$, $f = 1$. *(Tricky!)*

7 Estimating a Camera's Projection Matrix

In this section we consider the camera projection to be a 'black box'. That means that we do not care about the internal details. All we will assume here is that the relation

$$\begin{pmatrix} x_i \\ y_i \\ 1 \end{pmatrix} \sim \begin{pmatrix} \lambda_i x_i \\ \lambda_i y_i \\ \lambda_i \end{pmatrix} = M \begin{pmatrix} X_i \\ Y_i \\ Z_i \\ 1 \end{pmatrix}$$

maps *any* point in 3D space (X_i, Y_i, Z_i) to an image point (x_i, y_i) (of course, using the same (camera) matrix M for each point). This relationship (the *unknown* matrix M) can be calculated from examples.

Let the points (x_i, y_i) for $i = 1, \ldots, n$ be collected in a Matlab array `xy` such that `xy(i,1)` equals x_i and `xy(i,2)` equals y_i, *i.e.* the i-th row in the matrix `xy` is the point (x_i, y_i). Equivalently, let the corresponding 3D points (X_i, Y_i, Z_i) be collected in the matrix `XYZ`.

In Figure 8 an image of a calibration object is shown. The point correspondences for the 18 points marked with a '+' are available in the file `calibrationpoints.mat`. Loading this file (`load calibrationpoints`) will set the matrices `xy` and `XYZ`.

It may be of interest how we know those point correspondences: The extent of a calibration object (*e.g.* tile-size and angles between lines) is usually known to a very high precision. This means that, once the world coordinate system is chosen, we know the exact (world) location of certain points on the object. In this case we have chosen the world coordinate system such that its origin is on the bottom corner closest to the camera and the cartesian coordinate axes form a right-handed coordinate system aligned with the objects main axes (Z pointing "upwards", the positive octant lying "inside" the object).

The image coordinates are obtained by manually selecting the (image) points that the (world) points marked with a '+' are mapped to, using matlab function `ginput(18)`. (At least, that is how we do it for now. This step can be automated!)

You have seen the techniques to estimate a matrix M from point correspondences. The same principles apply, although now M is not a 3×3-matrix but a 3×4-matrix of the form

$$M = \begin{pmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \end{pmatrix}$$

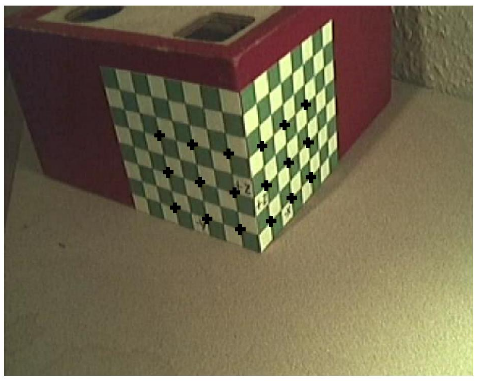

Figure 8: **Calibration object.** The calibration object consists of two perpendicular planes. On the planes square tiles are depicted (the edge-lengths of the tiles are taken to be 1). A convenient world coordinate frame is graphically hinted on the image. The image coordinates together with the corresponding 3D coordinates of the points indicated with a '+' are available in the file `calibrationpoints.mat`.

and can be rewritten by putting all the entries *row wise* into a vector

$$\mathbf{m} = (m_{11}, m_{12}, \ldots, m_{33}, m_{34}).$$

Note that using Matlab's function `reshape` to "linearize" a matrix by convention puts the entries of a matrix *column wise* into a vector. With those hints, you might think you should be able to do a determination of M on a minimal set of point correspondences. (How many do you need? How does that match the number of unknowns in M? Did you think of the scaling?)

However, we can easily specify more than just the minimal number of point correspondences, and hope to get a more accurate estimation of M using that redundancy. Fortunately, the methods we have used can be extended to redundant data, using standard linear algebra. Before, with the minimal number of correspondences, the equation $A\mathbf{m} = \mathbf{0}$ could be solved as the kernel of A, using the 'null()' command in Matlab.

Theory Questions

7.1. **(3 points)** Derive the matrix A for this problem, by following the usual steps (can I write down the equations the correspondence data gives, what are the unknowns, can I massage the equations into a matrix equation for the vector of unknowns).

7.2. **(3 points)** What happens to your matrix A if you have many more correspondences, can it still represent the resulting equations?

7.3. **(3 points)** Will there now be an exact solution to $A\mathbf{m} = \mathbf{0}$?

7.4. **(3 points)** A "best possible" solution for M can be found by minimizing $\|A\mathbf{m}\|$. Appendix B of the Lecture Notes shows how the SVD can help. Memorize the mantra ("last column of V in the SVD of A"), but understand each of the steps in appendix B! Explain in your own words how the procedure works.

So if the system is overdetermined, we can only find a “best possible” solution $\mathbf{m}$ to the equation $A\mathbf{m} = 0$, which is the one that *minimizes* $A\mathbf{m} \neq 0$. Appendix B derived that the solution is found by selecting the last column of the matrix V from the *singular value decomposition* of $A = UDV$. The Matlab code which does this is:

Listing 8: Singular Value Decomposition of the matrix A

```
% Do Singular Value Decomposition to obtain m
[U, D, V] = svd(A);
m = V(:, end);
% reshape m into the 3x4 projection matrix M
M = reshape(m, 3, 4)';
```

7.1 Questions and Exercises

⋆ **(15 points)** In analogy to what you did in Section 5, derive the Matrix A and write Matlab code that constructs it.

Hint: Copying and Pasting the code from the exercises in Section 5 does not give you the result, but it may provide you with a good framework to extend. Make sure to first derive the matrix (on paper) before you start programming!

⋆ **(15 points)** As Appendix B shows, you can find a solution for M by taking the last column of the matrix V from the singular value decomposition $A = UDV$ and reshaping it into a 3×4 matrix.

Write a function that does this. A possible signature for this function could be `function M = estimateProjectionMatrix(xy, XYZ)`.

8 Projecting Cubes into a Picture (Pseudo 3D)

Since now you know which projection matrix the camera uses, you can employ it to project your own virtual objects into the image as if they were present in the scene.

You will use simple cubes with side-length 1, aligned with the squares on the calibration pattern. This allows for an easy visual check of the projection matrix estimation.

You can use the given function `createCube` to create cubes, then you transform them using your own code, and you can draw them using the given function `subPlotFaces`.

8.1 Questions and Exercises

⋆ **(15 points)** Recreate Figure 9 by placing 3D cubes in the scene and projecting them into the image. Use the files `calibrationpoints.mat` and `calibrationpoints.jpg` available on the blackboard.

Hint: When placing the 3D cubes in the scene, keep in mind how the (world) coordinate system is defined.

• **(5 points)** Available on the blackboard there are different views on the calibration object (called `view1.jpg`, `view2.jpg`, `view3.jpg` and `view4.jpg`). Use those to obtain your own calibration points (using `ginput`), estimate the projection matrices and project the same cubes from above into the new images.

How many calibration points (per image) do you at least need?

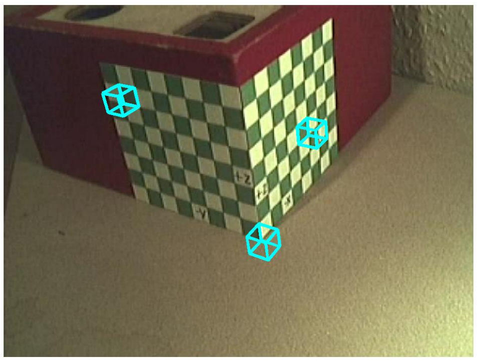

Figure 9: **Drawing 3D cubes in the scene.** The calibrated camera enables us to draw 3D objects in the scene.

> **Hint:** You may want to split the code into (at least) two functions, one to obtain the calibration points and one to perform the estimation/projection. Otherwise, you will have to click on all the calibration points, each time you test your code.

9 What to Hand In

Please refer to blackboard on how to hand in.
MAX POINTS: 200